IDUP and SPKM: Developing Public-Key-Based APIs and Mechanisms for Communication Security Services

Carlisle Adams

Bell-Northern Research
Ottawa, Canada
cadams@bnr.ca

Abstract

In this paper we discuss progress in the development of Application Program Interfaces (APIs) and mechanisms which provide a comprehensive set of security services to application developers. The APIs, though similar, are designed for distinct environments: the session API ("GSS") is aimed at the on-line real-time messaging environment; the store-and-forward API ("IDUP") is particularly suited to electronic-mail types of environments (where messages are secured independently of any on-line communication with intended recipients of those messages). Both APIs are designed to be easy to use, yet with appropriate public-key-based mechanisms (such as SPKM and PIM) include many necessary services for communication security, such as data origin authentication, data confidentiality, data integrity, and support for non-repudiation. A full key management and certification infrastructure can be provided by implementations of these APIs/mechanisms in a way which is completely transparent to the calling application, thus ensuring maximum flexibility and scalability to future environments.

1 Introduction

The increasing awareness of the need for security in virtually all forms of digital communication today has been coincident with the distressing realization by many application developers that security is not generally a simple "feature" which can be added into an existing product between one release and the next. In many cases, such "quick fixes" have severe security weaknesses due to the use of insecure algorithms, insecure protocols, or insecure implementation practices (such as placing cleartext keys in message headers, disk files, or easily visible areas of memory). Although relatively strong cryptographic algorithms and protocols can be found in the open literature, it is clear that what is needed is a straightforward way to make these tools available to application developers in a simple and easy-to-use manner. It is also clear from growing experience that hiding the often difficult and complex issues of key management from the calling applications (particularly in environments with very large user populations) would be of great benefit.

Application Program Interfaces (APIs) offering security services are quickly becoming recognized as the most practical way to meet these needs. They can provide a range of features to an application through a small number of API calls, thus ensuring high security without the substantial effort on the part of the application developer to implement the features "from scratch". Furthermore, the standardization of security APIs within such bodies as the Internet Engineering Task Force gives applications relatively transparent portability to any number of API implementation vendors (each offering its own set of security services and features).

This paper discusses the current status of two API development efforts which are on-going in the Common Authentication Technology (CAT) working group of the Internet Engineering Task Force (IETF). These APIs, a "session"-oriented API (Generic Security Service, GSS) and a "store-and-forward"-oriented API (Independent Data Unit Protection, IDUP), together with public-key-based underlying mechanisms, cover the full range of security services required by the vast majority of calling applications.

The outline of the remainder of this paper is as follows. Section 2 defines terminology: "APIs", "mechanisms", and "implementations" are concepts which are distinct but highly inter-related in this context and must be understood properly in order to clarify the remainder of the discussion. Section 3 describes the session API in terms of its main calls and features, along with a public-key-based mechanism which can be used to implement it. Section 4 describes the store-and-forward API, its main calls and features, and a public-key-based

mechanism which can be used to implement it. Section 5 compares the efforts described in the previous two sections with other work in this field. Section 6 closes the paper with concluding comments and observations.

2 APIs, Mechanisms, and Implementations

It is important to distinguish the concepts of "API", "mechanism", and "implementation" in order to make clear the discussion in the remainder of the paper (the usage here follows that employed in the IETF CAT group, among other places). "Application Program Interfaces" are purely interface specifications, defining the calls – and their corresponding parameters – which are available to the application developer. An API specification is typically accompanied by a document which gives the bindings to a specific programming language, detailing the relevant types, data structures, and values associated with all API calls, parameters, and status codes. It is common for some of the inputs and outputs of the API calls to be specified as nothing more than opaque binary buffers, called "tokens", whose contents need not be known in any way to the calling applications.

"Mechanisms", like APIs, are purely specifications, but have a different purpose and consequently a different level of detail. APIs specify in a generic way the security services which may be offered to an application and how to access them. Mechanisms, on the other hand, specify how the services are to be accomplished (and, if some optionality is available in the API, which options are to be made available). It is the mechanism specification which defines the content and format of all API "tokens", which typically contain security information (e.g., encrypted data) or security-related information (e.g., an authentication challenge or response). In the API, applications typically have an opportunity in an initialization type of call to select a particular underlying mechanism which will be used for subsequent calls; alternatively, a "default" value can be selected which will choose an environment-specific default mechanism for subsequent calls.

"Implementations", as would be expected, are the actual software or hardware modules which perform the security operations and services. An implementation must conform to the API specification and include one or more mechanisms. Software implementations will typically be made available as run-time libraries which can be linked into applications making API-conformant calls. The resulting applications can apply the offered security services to their communications by sending and receiving API "tokens" (rather than raw data) in their message exchanges.

2.1 API Philosophy

There are two main things to note about the definition and usage of APIs in this context. The first is about portability and the second is about data transfer. The intention is that an application coding to an API specification (and using "default" parameters wherever applicable) should be portable across all implementations of all underlying mechanisms of that API. To the extent that a mechanism goes beyond the API in defining new features or calls and an application makes use of those extensions, that application will not be transparently portable to other mechanisms. Furthermore, an application which selects a particular mechanism to provide its security services should be portable across all implementations of that mechanism. Again, an implementation which goes beyond the mechanism specification will render non-portable any application which makes use of those extensions. As an aid to application portability, then, implementations should conform to mechanism specifications and mechanisms should conform to API specifications, as much as is possible.

Secondly, in the interest of creating generic specifications, APIs (and therefore mechanisms and implementations) typically remain completely separate from data transfer. That is, API implementations typically do not include such things as communication stacks and transport- and network-layer protocols for the purpose of transmitting and receiving application data (note, however, that they may include such things for their own purposes, completely transparent to the calling application). The explicit assumption stated in typical API specifications is that the calling applications are themselves communicating entities (or, at least know how to get data to and from a communications repository such as an ftp site) which use the API to protect data which they are about to send and "unprotect" data which they have just received. The API calls, then, do not replace an application's communications calls; they are simply logically inserted before the Send() call and after the Receive() call in the application code.

2.2 IETF Security Standardization Status

Typical communication requirements tend to fall into one of two possible environments: on-line / session-oriented / real-time types of environments; and off-line / e-mail-oriented / store-and-forward types of environments (note that in a broad sense, archiving applications are included in this latter class). Although it is possible to add security to both types of environments using only symmetric-key cryptographic schemes (e.g., schemes based on the Data Encryption Standard), some applications require – because of convenience, scalability, or non-repudiation considerations – an approach based on a combination of symmetric and asymmetric (i.e., public-

key) techniques, with reliance upon a suitably-deployed public-key infrastructure. The discussion in the remainder of this paper is primarily focused on API/mechanism support for such applications.

Until recently, the standardization status in the IETF regarding security work revealed an interesting dichotomy. For the session environments, RFC-1508 and RFC-1509 define the Generic Security Service (GSS) API [12] and associated C-language bindings [17]. This API is widely considered to be comprehensive and useful, offering a good range of security services to application developers. However, the only public-key-based GSS-API mechanism defined was the Distributed Authentication Security Service (DASS) [8], which was offered as an "Experimental Protocol" (as opposed to a Proposed Standard) and, in some ways, did not use the full potential of its public-key infrastructure (for example, the "signatures" were not true digital signatures, but checksums computed using a DES-MAC). A need was still felt for a true, public-key-based GSS-API mechanism.

For the store-and-forward environments, the opposite situation had occurred. Work had been done on so-called "enveloping protocols", perhaps the most noteworthy of which being RFC-1421, the Internet Privacy Enhanced Mail (PEM) Proposed Standard, but later offerings including MIME [3] and MOSS [4], among others. Viewed from a certain perspective (perhaps with the benefit of hindsight), it seems clear that these proposals could be regarded as public-key-based mechanism specifications in need of a store-and-forward API.

Thus, in an attempt to complete the "portfolio" of tools in the security standardization work within the IETF, the CAT group has spent considerable effort developing and advancing both a public-key-based GSS-API mechanism (the Simple Public-Key GSS-API Mechanism, [1]) and a generic store-and-forward API (the Independent Data Unit Protection GSS-API, [2, 5]), along with a specification which transforms the PEM RFC into an IDUP-conformant mechanism [6]. A brief description of this recent CAT work is included in the following two sections.

3 Session-Oriented Security Activities

As stated above, within the IETF the session-oriented API is the Generic Security Service (GSS) API. It defines calls to initiate, accept, and delete a "security context" – a secure (logical) connection between two communicating entities – and to sign/verify and seal/unseal data sent across this context. "Sealing" data encapsulates it into a token, providing cryptographic integrity and optionally encrypting it (if the calling application has requested this). "Signing" data provides cryptographic integrity for it, but does not encapsulate or encrypt the data. [Note that "gss_sign()" and "gss_seal()" have now been renamed to "gss_getMIC()" and "gss_wrap()", so that the corresponding SPKM tokens described below are called SPKM_MIC and SPKM_WRAP, respectively.]

Perhaps the best-known GSS-API underlying mechanism is one which has been designed based on the Kerberos Network Authentication Service (V5) [10, 14]. This Kerberos-based mechanism [13], currently an Internet Draft, is expected to be progressed to Proposed Standard status in the near future.

The primary purpose behind the development of the Simple Public-Key GSS-API Mechanism (SPKM) was to provide a GSS-compliant underlying mechanism which was based on a public-key infrastructure but was as similar to the Kerberos 5 mechanism as possible. This is because the Kerberos 5 GSS mechanism [13] is beginning to gain wide acceptance within the Internet community (and so a familiar, drop-in replacement would be desirable), but its reliance on a symmetric cipher infrastructure (DES) means that it cannot support a non-repudiation service and that secure timestamps must be available in the environment in which it is used. Furthermore, issues such as scalability to communities of tens of thousands of users (or more) become much more manageable using a public-key infrastructure than a symmetric-key infrastructure.

3.1 SPKM Features

The Simple Public-Key GSS-API Mechanism was specifically designed to provide the following features, which are not readily available in other existing GSS mechanisms.

1 SPKM allows three-pass mutual authentication, whereas other mechanisms use only two-pass mutual authentication; this enables environments which do not have access to secure timestamps to nevertheless have access to secure mutual authentication.

2 SPKM uses Algorithm Identifiers to specify various algorithms to be used by the communicating entities. This allows maximum flexibility for a variety of environments, for future enhancements, and for alternative algorithms.

3 SPKM allows the option of a true, asymmetric-algorithm-based, digital signature in the gss_sign() and gss_seal() operations, whereas other mechanisms use an integrity checksum based on a MAC computed with a symmetric algorithm (DES). For some environments, the availability of true digital signatures which can support a non-repudiation service is a necessity.

4 The key management employed in SPKM is intended to be as compatible as possible with both X.509 [7] and PEM [9], since these represent large communities of interest and show relative maturity in standards.

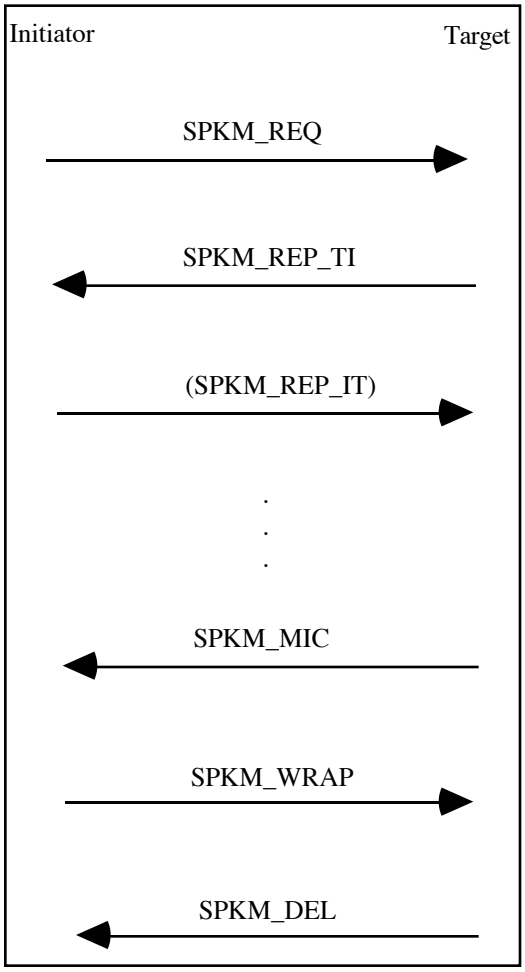
5 SPKM supports the ability of the initiator to remain anonymous while the target is authenticated, a desirable (or necessary) authentication option in some environments.

6 During context establishment in SPKM, full negotiation of confidentiality algorithms, integrity algorithms, and key establishment algorithms is possible between the communicating entities. The resulting context may support any number of mutually-agreed confidentiality and integrity algorithms, which may be selected by the calling application on a per-message basis, depending on the security needs of the individual messages.

Further to point (6) above, SPKM allows the context initiator and target to negotiate any standard mechanism for key establishment. For example, if one-way authentication is used, the mechanism may be an RSA key transport mechanism; on the other hand, if mutual authentication is used, the mechanism may be Diffie-Hellman (which allows both parties to participate in the creation of the context key).

As well, the negotiation of a set of symmetric and integrity algorithms which may be used over the context is a very important feature for an environment such as the Internet where specific algorithms cannot be guaranteed to be available to implementations in other countries due to export limitations, patent restrictions, or a variety of other reasons. Such a negotiation at context establishment time allows the SPKM mechanisms at both ends to agree upon a set of underlying algorithms in a way which is completely transparent to the calling applications. It is important to provide this transparency in order to maintain the intent of GSS itself (that of supplying a set of security services to an application without requiring that the application have any special knowledge of security algorithms, protocols, or procedures).

The following diagram illustrates the random-number-based SPKM protocol exchanges which may take place during the establishment and use of a GSS security context.



In the diagram, the context initiator calls gss_init_sec_context() and receives the token SPKM_REQ as a return parameter, which it then sends to the intended recipient (the context acceptor, or target) using any convenient communications protocol. The target, upon receiving this token, inputs it to gss_accept_sec_context() and receives the token SPKM_REP_TI as a return parameter, which it then sends back to the initiator. The initiator inputs this token to a second call to gss_init_sec_context(). If unilateral authentication was initially chosen by the initiator, context establishment is complete at this point. Otherwise (i.e., if mutual authentication was initially requested) SPKM_REP_IT is returned by gss_init_sec_context() and is sent to the target, who inputs it to a second call to gss_accept_sec_context(), thereby completing context establishment. At this point, SPKM_MIC and SPKM_WRAP tokens can be freely exchanged by both parties until an SPKM_DEL token is sent by either party to delete (i.e., shut down) the context.

The token SPKM_REQ contains, among other data fields, the following items:

- the initiator and target names;
- a "fresh" random number;
- a list of available confidentiality algorithms;
- a list of available integrity algorithms;
- a list of available key establishment algorithms;
- a context key (or key half) corresponding to the first key estb. alg. given in the previous list;
- GSS context options/choices (such as unilateral or mutual authentication, use of sequencing and replay detection, and so on).

The above information is integrity protected and authenticated by the initiator's digital signature (unless the initiator wishes to remain anonymous, in which case the information is integrity protected only using a MAC keyed with the offered context key, and the first item above uses a suitable indication of "anonymous" for the initiator's name). Public-key certificates and full certification path information may optionally be sent as part of the SPKM_REQ token, if necessary.

The token SPKM_REP_TI contains, among other data fields, the following items:

- the initiator and target names;
- the random number sent by the initiator;
- a "fresh" random number;
- the subset of offered confidentiality algorithms which are supported by the target;
- the subset of offered integrity algorithms which are supported by the target;
- an alternative key establishment algorithm (chosen from the offered list) if the first one offered is unsuitable;
- the key half corresponding to the initiator's key estb. alg. (if necessary), or a context key (or key half) corresponding to the key estb. alg. given above;
- GSS context options/confirmations (such as unilateral or mutual authentication, use of sequencing and replay detection, and so on).

The above information is integrity protected and authenticated by the target's digital signature. Note that the target's signature on the initiator's "fresh" random number gives the initiator the assurance that the target is "live" and authentic. Public-key certificates and full certification path information may optionally be sent as part of the SPKM_REP_TI token, if this was requested by the initiator in SPKM_REQ.

If mutual authentication has been requested, then the token SPKM_REP_IT is generated by the initiator. It contains, among other data fields, the following information:

- the initiator and target names;
- the random number sent by the initiator in SPKM_REQ;
- the random number sent by the target in SPKM_REP_TI;
- the key half corresponding to the target's key estb. alg. (if necessary).

The above information is integrity protected and authenticated by the initiator's digital signature. Note that the initiator's signature on the target's "fresh" random number (from SPKM_REP_TI) gives the target the assurance that the initiator is "live" and authentic.

At the completion of random-number-based context establishment, therefore, one or both sides have been authenticated (without the use of secure time), a context key has been securely established, and two sets of algorithms (confidentiality and integrity) have been agreed upon for use over the context for the purposes of MIC and WRAP operations. Each SPKM_MIC and SPKM_WRAP token, as well as the final SPKM_DEL token, can explicitly specify relevant algorithms from the agreed sets to be used for the data associated with that token, or may simply specify "default" algorithms (the first algorithm in the each of the agreed lists). The tokens therefore contain, among other data fields, the following information:

- the integrity algorithm, or "default";
- the confidentiality alg., or "default" (Wrap only);
- the sequence number (if used over context);
- the integrity checksum (computed over the data and the above info. using the int. alg. specified above);
- the input data (Wrap only).

The above format gives great flexibility to the calling applications to select a "Quality of Protection" which is appropriate for the data being protected within a single security context.

4 Store-And-Forward-Oriented Security Activities

Within the IETF, a great deal of effort was put forth on the development and adoption of the Internet Privacy Enhanced Mail (PEM) enveloping protocol [11]. Since its progression to Proposed Standard, work has begun on other enveloping protocols both within and outside the IETF, including Multi-part Internet Mail Extensions (MIME) [3], MIME Object Security Services [4], Secure MIME [15], and the Message Security Protocol [16].

As stated above, until recently all the enveloping protocol alternatives listed in the previous paragraph could properly be viewed as candidates for mechanisms underlying a yet-to-be-defined API for store-and-forward

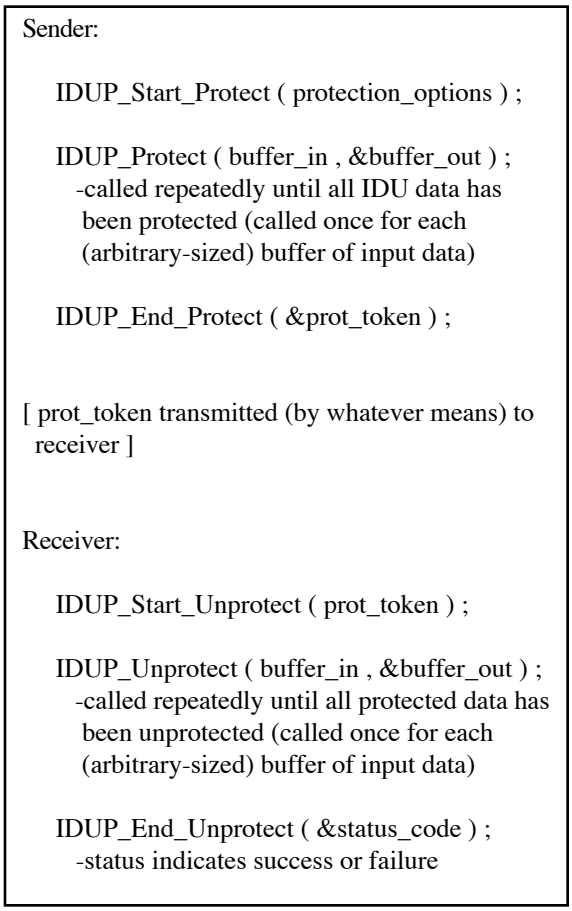
environments. It seemed clear that such an API would therefore not only serve to unify this work, but also create a standardized framework to which application developers could code in order to invoke desired security services. The specification of this framework, then, was the motivation behind the IDUP-GSS-API.

The Independent Data Unit Protection (IDUP) specification is meant to be an extension to the GSS-API for store-and-forward environments, re-using as much as possible of the GSS-API's structure and calls. One of the primary differences from GSS is that in the store-and-forward environment no real-time security context is established between the communicating entities and no entity authentication is done prior to the transfer of data from sender to receiver. The data to be transferred (or stored, in an archiving environment) is seen as an IDU, an "Independent Data Unit", which must be secured independently of any other data which may be transferred and independently of any on-line communication with intended recipients of the data.

In such an environment, data origin authentication is of great importance, as is some assurance that the intended receiver is the only one who will view any encrypted contents. The secured data which results from calls to IDUP can be made available to the recipient in any convenient way (for example, it may be attached to or embedded in an e-mail message, it may be placed on an unsecured file server to be downloaded using ftp, or it may be archived to a tape to be retrieved only months or years later).

4.1 IDUP Calls and Features

The number of calls which are unique to IDUP (as opposed to those which are in common with the GSS-API) is relatively small. The main concept in IDUP is that of "protecting" and "unprotecting" the IDU; thus there are "protection" and "unprotection" sets of calls. Each set is comprised of three parts, so that there is a Start_Protect() call, a Protect() call, and an End_Protect() call (likewise for "unprotect"). The reason for this choice is that in many store-and-forward environments the IDU to be secured may be fairly large in size (larger than convenient buffer sizes on the computing platform). Such an arrangement of calls allows the application to choose whatever buffer size is most appropriate, call Start_Protect(), and then call Protect() as many times as is necessary, handing in a buffer at a time and getting back a buffer of (possibly encrypted) output data, if available. Once the application has protected all IDU buffers, it calls End_Protect() to complete the operation and to receive, as output, a token ("prot_token") which can be used by a legitimate receiver of the data for the unprotection set of calls. This receiver begins with Start_Unprotect(), calls Unprotect() as many times as necessary for whatever buffer size it has chosen, and completes the operation with End_Unprotect(), receiving a flag indicating whether or not the operation was successful, along with information about the originator (the entity who protected the IDU), if the underlying mechanism supports this.



The IDUP shares some features with the GSS-API, such as relative ease of use and the flexibility to choose options and security services which match the needs of a particular calling application or piece of data. Because its primary purpose is for use in store-and-forward types of environments, however, some features are unique to IDUP. For example, IDUP incorporates a very general framework for the generation and processing of "evidence" the "proofs of" which are needed by some applications including such things as proof of origin, proof of delivery, and proof of receipt (i.e., those proofs which may be required for the provision of full non-repudiation services). Thus, for example, a calling application may stipulate that it wants a proof of receipt when it protects an IDU and specify which target(s) it wants a receipt from. The named recipient(s), when unprotecting the data, will be notified of this request by a flag and will then have the opportunity to generate the receipt and send it back to the originator.

4.2 An Example Mechanism

Having made the assumption that much of the work which had been done in the IETF and elsewhere on enveloping protocols could be regarded as candidate mechanisms under an appropriately-defined store-and-forward API, it was necessary to support this conjecture with a concrete example. Thus the "PEM-Based IDUP Mechanism (PIM)" has been put forward as an Internet Draft within the CAT working group [6]. PEM was chosen first partly because of its familiarity and partly because of its status as a Proposed Standard, but it is felt that IDUP mechanisms based on MOSS, S/MIME, or MSP can and likely will be specified in the near future.

PIM is essentially a mapping of RFC-1421 to the IDUP calls, specifying the contents of prot_token (virtually identical to the PEM header), the security options not supported (such as evidence generation/processing), and the actions which must be undertaken by the calling application in order to create fully PEM-compliant messages (such as canonicalization of the input data and PEM encoding of the (possibly encrypted) output data). It is anticipated that the corresponding activities for other enveloping protocols will be similarly straightforward.

5 Comparisons with Other Work

5.1 SPKM

The principle features of SPKM which set it apart from other existing GSS-API mechanisms are listed at the beginning of Section 3.1 (such as true digital signatures, algorithm negotiation, availability of initiator anonymity, and the non-reliance on secure time). However, a slightly more detailed comparison with Kerberos [13] may be instructive. The primary difference between a public-key-based mechanism and a symmetric-key-based mechanism is the infrastructure (the administrative processes which must be on-line for the system to work). In Kerberos, one or more Authentication Servers (AS) must be on-line. These servers must run on physically secure computing platforms (hosts) primarily because they each contain a database of principals (i.e., users and servers) and their secret keys; any compromise of these servers can lead to a compromise of the entire system.

In a public-key-based mechanism such as SPKM, all that needs to be on-line is an unprotected database of public-key certificates. A Certification Authority is necessary at system initialization time to create these certificates, but it need not be perpetually on-line subsequent to this. Furthermore, if the database is replicated (as is specified for the X.500 directory system, for example), then no single point of compromise exists which can compromise the entire system – physical security is no longer a necessary condition for the security of the system.

Understanding the difference in infrastructure between SPKM and Kerberos is useful for understanding how other concepts compare. For example, a *realm* or "administrative domain" in Kerberos is determined in a public-key-based mechanism by the number and inter-relationship of the various Certification Authorities in the system. A CA domain essentially defines a realm; inter-realm operation is accomplished through multi-CA cross-certification. *Delegation* can be supported if a user's private key can be used to sign a randomly-generated, short-lived public key (i.e., to certify a key pair created for delegation purposes only). This involves domain policy considerations and other complexities, however, and is often not available in public-key environments. Consequently, there are no known implementations of SPKM which currently support delegation (this area is for further study). *Tickets* and *ticket-granting tickets* in Kerberos are unnecessary in a public-key infrastructure because Authentication Servers do not exist – authenticated communications are always established directly with the intended destination without the need for an intermediary to provide tickets for subsequent communications. Finally, although the details differ, the process of *user creation/initialization* is similar in both systems: the user must, by some out-of-band means, enter into a (physical or electronic) communication with the system authority (the AS or the CA) in which the user establishes his/her system identity and all necessary secret information is securely transferred between the authority and the user.

SPKM also differs from the Distributed Authentication Security Service (DASS) [8] in that it does not rely on secure timestamps for replay detection during entity authentication, and it makes full use of its underlying public-key infrastructure (e.g., for true digital signatures during the sign and seal operations). These may be viewed as important or necessary features in some environments.

5.2 IDUP

As mentioned in Section 4, IDUP was created because the working groups within the IETF were unaware of any other store-and-forward APIs available to consolidate the many enveloping protocols being proposed. It is therefore difficult to compare IDUP directly with other work. However, once a number of IDUP mechanisms have been specified (PEM-based, MOSS-based, MSP-based, and so on), it will be interesting to compare these to see their relative advantages and disadvantages.

6 Concluding Comments

The Application Program Interfaces and mechanisms described in this paper are a significant step toward the goal of providing a set of necessary security services to application programmers in a way which is flexible, standards-based, and easy to use. Both Generic Security Service (GSS, the on-line API) and Independent Data Unit Protection (IDUP, the store-and-forward API), with appropriate underlying mechanisms such as SPKM and PIM, offer data origin authentication, data confidentiality, data integrity, and support for non-repudiation to applications, services which are essential for distributed network security over insecure and potentially hostile environments, including the Internet.

Both the GSS/SPKM and IDUP/PIM specifications are also able to hide all aspects of key generation, key distribution, and key management from the calling application (and therefore from the application user). In both cases the application needs to be aware of whatever information is required to find and "unlock" a repository of opaque credential information for its user, but need not know anything about symmetric keys, public/private keys, session keys, or any of the underlying algorithms and protocols. This is in keeping with the design philosophy of security interface specifications, which strives to allow mechanism implementors to build something which provides security without imposing undue constraints on the calling applications (the applications simply make a few calls to properly secure their data and otherwise need to know virtually nothing about how the security is actually provided).

It is felt that the APIs and mechanisms described in this paper and being standardized by the Internet Engineering Task Force will prove to be a useful way to provide distributed network security to a variety of applications, including electronic mail packages, database management systems, telnet packages, and so on. In this way, secure e-mail, secure DBMSs, secure telnet, and a host of other secure applications can be developed with minimal time and effort on the part of the application developers.

References

[1] C. Adams, "The Simple Public-Key GSS-API Mechanism (SPKM)", Internet Draft draft-ietf-cat-spkmgss-04.txt (work in progress).

[2] C. Adams, "Independent Data Unit Protection Generic Security Service Application Program Interface", Internet Draft draft-ietf-cat-idup-gss-02.txt (work in progress).

[3] N. Borenstein, N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies".

[4] S. Crocker, N. Freed, J. Galvin, S. Murphy, "MIME Object Security Services", Internet Draft draft-ietf-pem-mime-08.txt (work in progress).

[5] D. Grebovich, "Independent Data Unit Protection Generic Security Service Application Program Interface: C-bindings", Internet Draft draft-ietf-cat-idup-cbind-01.txt (work in progress).

[6] D. Grebovich, C. Adams, "PEM-Based IDUP Mechanism (PIM)", Internet Draft draft-ietf-cat-pim-00.txt (work in progress).

[7] ISO/IEC 9594-8, "Information Technology - Open Systems Interconnection - The Directory: Authentication Framework", CCITT Recommendation X.509.

[8] C. Kaufman, "DASS: Distributed Authentication Security Service".

[9] S. Kent, "Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management".

[10] J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5)".

[11] J. Linn, "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures".

[12] J. Linn, "Generic Security Service Application Program Interface".

[13] J. Linn, "The Kerberos Version 5 GSS-API Mechanism", Internet Draft draft-ietf-cat-kerb5gss-02.txt (work in progress).

[14] B. C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks", IEEE Communications, vol.32 #9, Sept. 1994, pp.33-38.

[15] PKCS Steering Group (RSA Data Security, Inc.), "PKCS Security Services for MIME ("S/MIME")", Preliminary Draft document, April 19, 1995.

[16] U.S. National Security Agency, "Message Security Protocol", Secure Data Network System SDN.701, March 1994.

[17] J. Wray, "Generic Security Service Application Program Interface: C-bindings".